3
Message Digests

In this chapter, we will learn about **message digests**, also known as **cryptographic hashes**. Message digests, which are produced by cryptographic hash functions, have many important applications, such as data integrity checking, password verification, digital signatures, secure network protocols such as TLS, and even blockchains. We will provide an overview of the cryptographic hash functions supported by **OpenSSL** and recommendations on which hash functions to use and which to avoid. Then, in the practical part of this chapter, we will learn how to calculate a message digest using the command line and C code. After reading this chapter and trying the practical part, you will know why message digests are needed and how to calculate them.

We are going to cover the following topics in this chapter:

- What are message digests and cryptographic hash functions?
- Why are message digests needed?
- Assessing the security of cryptographic hash functions
- Overview of the cryptographic hash functions supported by OpenSSL
- How to calculate a message digest on the command line
- How to calculate a message digest programmatically

Technical requirements

This chapter will contain commands that you can run on a command line and C code that you can compile and run. For the command-line commands, you will need the `openssl` command-line tool, along with the relevant OpenSSL dynamic libraries. To build the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement an example program in this chapter, in order to practice what we are learning. The full source code of that program can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter03`.

What are message digests and cryptographic hash functions?

A **message** in cryptography is any piece of data, big or small, that is processed by a **cryptographic algorithm**.

A **cryptographic hash function** is an algorithm that maps a message of arbitrary size to a relatively short (for example, 256 bits) fixed-size array of bits. This fixed-size bit array is called a **message digest** or a **cryptographic hash**.

In other words, a message digest is the output of a cryptographic hash function. As we mentioned in the previous chapter, we can say that a message digest is a cryptographically strong checksum.

A good cryptographic hash function has the following properties:

- It is *deterministic*, meaning that processing the same message must always yield the same message digest.
- It is *irreversible*, meaning that it must be impossible or extremely difficult to recover the original message by its digest. The only way to reverse the hash should be *brute force* and it must be too computationally expensive to do.
- It must be computationally infeasible to find two distinct messages with the same digest. If two or more different messages with the same digest are found, then this is known as a **hash collision** or simply a **collision**.
- Any change to the message, big or small, must result in an extensive change to the digest – so extensive that it would be impossible to relate the two digests to each other. Such an extensive reaction, even to a small change, is called the **avalanche effect**.

It's not difficult to understand what message digests are, but *why are they needed*, and *why are they important?*

Why are message digests needed?

Message digests have many applications. The most obvious one is **data integrity verification**.

Data integrity verification

When you're downloading software from the internet, you will often find the message digest of the distribution package close to the download link. As an example, look at the OpenSSL download page at `https://www.openssl.org/source/`:

KBytes	Date	File
14627	2021-Sep-07 12:00:26	openssl-3.0.0.tar.gz (SHA256) (PGP sign) (SHA1)
9603	2021-Aug-24 13:46:31	openssl-1.1.1l.tar.gz (SHA256) (PGP sign) (SHA1)
1457	2017-May-24 18:01:01	openssl-fips-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)
1437	2017-May-24 18:01:01	openssl-fips-ecp-2.0.16.tar.gz (SHA256) (PGP sign) (SHA1)

Figure 3.1 – The OpenSSL download page fragment

Next to the download link, you will see links called `SHA256` and `SHA1`. These contain the message digests for the corresponding `.tar.gz` files. When the required `.tar.gz` file is downloaded, the digest of the downloaded file can be calculated and compared to the expected digest. If the digests match, then the file may not have been corrupted or otherwise changed while being transmitted and saved. If the file was changed, the calculated digest would look very different from the expected digest, because of the *avalanche effect*. In our case, the OpenSSL distribution package is a rather small download, and the official OpenSSL website is a trusted source, so the chance of the download being corrupted is very small. But imagine if you were downloading something big, such as a Linux installation image, or you received a distribution from an unofficial source and wanted to check that it's not been tampered with. Here, you can obtain the correct digest from the official website and check if the digest of the untrusted download matches.

Message digests are also applicable to stored data. If some data is stored for a long time, it may be corrupted because of storage media errors or other unexpected changes. Verification via message digest is a reliable way to check that the data has not been corrupted.

Basis for HMAC

Message digests form the basis for **Hash-based Message Authentication Codes** (**HMACs**). HMAC is a **Message Authentication Code** (**MAC**) that combines a secret key and a cryptographic hash function for data authentication. HMAC will be covered in more detail in *Chapter 4*, *MAC and HMAC*.

Digital signatures

When a message is digitally signed, the digest of that message is signed. It is worth mentioning that digital signatures are used not only for documents. They are often used for signing drivers, applications, and even cryptography objects, such as the **X.509 certificates** that are used in the TLS protocol. Digital signatures will be covered in more detail in *Chapter 7*, *Digital Signatures and Their Verification*.

Network protocols

Security-enabled network protocols, such as TLS and SSH, use the HMACs and digital signatures of transmitted data at different protocol stages. As we mentioned previously, cryptographic hash functions are needed for HMAC and digital signatures, so they are also needed for secure network protocols.

Password verification

Most modern operating systems and websites do not store passwords in plaintext. Often, a salted password hash is stored instead. Salted means that some salt is added to the password before hashing. Salt is a small amount of random data, which can be as small as 2 bytes, that is prepended or appended to the password so that the same passwords will have different hashes and be more difficult to break with the help of a known hash. The salt itself is not a secret and is usually stored together with the hash. When a password is verified, it is concatenated with the stored salt and hashed. Verification is successful if the computed digest is the same as the one that was stored.

Content identifier

As one of the features of a good cryptographic hash function is an extremely low probability of a hash collision, message digests can be used for identifying objects. Let's look at some examples of how cryptographic hashes are used for identification.

When verifying an X.509 certificate, the issuer certificate is looked up by its name and the cryptographic hash of its public key.

Modern **Source Code Management** (**SCM**) systems, such as Git and Mercurial, use cryptographic hashes to uniquely identify stored objects, such as files, commits, branches, and tags. Such an identification scheme also ensures the integrity of the code repository.

Some peer-to-peer filesharing networks, such as eDonkey, use message digests to identify shared files.

Blockchain and cryptocurrencies

Blockchain blocks are linked together using cryptographic hashes. Newer blocks refer to the old blocks by their hashes. Such referencing ensures the integrity of the chain and the impossibility to change older blocks since any block change will also change its hash. Hence, its identity will also change, and the changed block will stop being part of the blockchain.

The most well-known application of blockchain technology is **cryptocurrencies**. But it's not the only one. The aforementioned **Git** and **Mercurial** SCMs use the same kind of link between their stored objects, so they are also based on blockchain technology. There are also many other uses of blockchain: in trading, medicine, and other areas. The blockchain would be impossible without cryptographic hash functions and message digests.

Proof-of-work

A **proof-of-work** system gives its client a cryptographic challenge that can only be overcome if the client performs a certain amount of computational work. One such challenge is **partial hash inversion**, meaning that the client has to find some piece of data whose partial message digest is the same as requested in the challenge. If a good cryptographic hash function is used, the client has to perform some brute force work. The data that's found will be proof that the client performed the work.

Proof-of-work systems can be used for cryptocurrency mining, processing a cryptocurrency transaction, and preventing **denial-of-service** attacks or **spam**.

Let's explain this concept using the spam prevention example. Imagine that a mail client connects to a mail server and wants to send an email to one or several addresses. The server gives the client proof-of-work challenges – one for every target address. If the client wants to send the message to one or only a few addresses, it will need to spend only a small volume of computational resources, which will not be a big burden. However, if a spammer wants to send the same mail to a million addresses, the volume of computational work will be unbearable.

The important property of a proof-of-work system is that it is much easier to check the performed work than to perform it.

Why does only a partial hash have to be found in a proof-of-work challenge? Because reverting the full hash of a good cryptographic hash function is too expensive and can take billions of years. The purpose of a proof-of-work challenge is to prevent service abuse, not to block the service completely.

As we can see, cryptographic hash functions and message digests are very important and have a lot of good appliances. But cryptographic hash functions are only useful if they are secure. Let's learn about the popular attacks that are performed on message digest functions, how can we measure their security, and how much security is enough for a hash function.

Assessing the security of cryptographic hash functions

The two most important types of attacks that can be performed on cryptographic hash functions are collision attacks and preimage attacks. A collision attack tries to find two messages that produce the same message digest, while a preimage attack aims to find a message that produces the predefined message digest. Performing a collision attack is easier than performing a preimage attack on the same hash function because the message search is not limited to a single target message digest.

The security level of a cryptographic hash function is the computational complexity of the collision attack. The security level is measured in security bits, similar to the security level of symmetric encryption algorithms. For example, if the complexity of the collision attack is 2,128 hash evaluations, then the security level is 128 bits.

The security level of a cryptographic hash function depends on the size of its output message digest. If the message digest's size is n bits, the maximum attack complexity is 2n/2 for the collision attack and 2n for the preimage attack. It is impossible to have a higher complexity than 2n/2 for the collision attack because the birthday attack, which is based on the birthday paradox, can always find collisions in 2n/2 time.

For example, the **SHA-256** function has a 256-bit hash size, a 2^{128} collision attack complexity, a 2^{256} preimage attack complexity, and a security level of 128 bits. It is also said that the function's collision resistance level is 128 bits and that its preimage resistance level is 256 bits.

How many security bits is enough? The same as for symmetric encryption algorithms:

- 112 bits of security should be enough until 2030.
- 128 bits of security should be enough until the next revolutionary breakthrough in technology or mathematics.

So far, we know how hash function security is measured and how much security is enough. Next, we will learn about which cryptographic hash functions are supported by OpenSSL and how secure they are.

Overview of the cryptographic hash functions supported by OpenSSL

In this section, we will review the cryptographic hash functions supported by OpenSSL and their properties, such as security, speed, and message digest size. We will also learn a bit about the history of those hash functions.

Reviewing the SHA-2 family of hash functions

The **SHA-2** family contains the most popular modern cryptographic hash function, SHA-256. SHA-256 outputs a 256-bit digest and has a collision resistance level of 128 bits.

SHA-256 is widely used. It is currently the default hash function that's used in the TLS protocol, as well as the default signing function for X.509 certificates and SSH keys. Several cryptocurrencies, including Bitcoin, use SHA-256 to verify transactions and proof of work. The popular Git SCM is migrating to SHA-256 hashes for its blockchain implementation and object identification processes. SHA-256 is used in many security protocols and software, such as SSH, IPsec, DNSSEC, PGP, and more. SHA-256 is also a popular choice for password hashing in Unix systems.

In addition to SHA-256, the SHA-2 family also includes the following hash functions:

- SHA-224, which is a modification of SHA-256 with a changed *initialization vector* and *truncated output* (224 bits). Its collision resistance is 112 bits.
- SHA-512, which has an algorithm similar to SHA-256 but operates on 64-bit words instead of 32-bit words. It outputs a 512-bis message digest and has a collision resistance of 256 bits.
- SHA-384, SHA-512/256, and SHA-512/224, which are modifications of SHA-512 with changed initialization vectors and truncated output. The functions output message digests of 384, 256, and 224 bits and have collision resistances of 192, 128, and 112 bits, respectively.

SHA-2 functions were developed by the United States **National Security Agency** (**NSA**) and first published by the **National Institute of Standards and Technology** (**NIST**) in 2001 as a federal standard. SHA-2 algorithms are patented, but the patent is available under a royalty-free license.

SHA stands for **Secure Hash Algorithm**. SHA-2 functions are currently considered secure, but cryptanalysts are slowly and steadily advancing their attacks on SHA-2. While the attacks are advancing, there are currently no successful attacks on full versions of SHA-2, only on versions with reduced amounts of rounds. Therefore, currently, all SHA-2 functions have a collision resistance level equal to the amount of hash bits divided by 2, which is the maximum possible.

SHA-256 and its modifications operate on *32-bit words*, while **SHA-512** and its modifications operate on *64-bit words*. Because of that, SHA-256 calculation is 2-4 times faster than SHA-512 on 32-bit CPUs, but SHA-512 is approximately 1.5 times faster than SHA-256 on 64-bit CPUs.

SHA-2 functions have an acceptable speed for modern hardware but are slower than their predecessor, the SHA-1 function. For example, the SHA-256 function is two times slower than SHA-1.

Modern x86_64 CPUs from Intel and AMD support **Intel SHA extensions**, which speed up the process of calculating SHA-256 and SHA-1. Some ARMv8 CPUs, such as Cortex A53, support **ARMv8 Cryptography Extensions**, which also speed up of the process of calculating SHA-256 and SHA-1, as well as SHA-224.

At the time of writing, SHA-2 functions are the most popular cryptographic hash functions, but the *SHA-2 family* already has a successor: the *SHA-3 family*.

Reviewing the SHA-3 family of hash functions

Unlike the in-house development of SHA-2, the **SHA-3** algorithms were chosen through an algorithm competition. This is similar to how the **Advanced Encryption Standard** (**AES**) algorithm was chosen through another competition. NIST organized such a competition because there were successful attacks on SHA-2's predecessors, namely **SHA-1**, **SHA-0**, and **MD5**, which were built on the same principles as SHA-2; there was a need for a hash function based on different principles. SHA-3 selection was a long process. NIST started to organize the competition in 2006 and officially announced it in 2007. The applicants submitted their algorithms until 2009. Then, there were two competition rounds in 2009 and 2010. After that, the winner was chosen in 2012, the standard was published in 2014, and finally it was approved in 2015.

Note that SHA-2 algorithms are not being deprecated or abandoned because of the standardization of SHA-3. The purpose of SHA-3 is to expand the diversity of good message digest algorithms and to provide a ready and easy replacement for SHA-2 if it will be needed.

The winner of the SHA-3 competition was the **Keccak** algorithm from a team of Belgian cryptographers. One of those cryptographers was Joan Daemen, who is also the co-author of the **Rijndael** algorithm (AES).

The SHA3 algorithm family consists of the following functions:

- SHA3-224
- SHA3-256

- SHA3-384
- SHA3-512
- SHAKE128
- SHAKE256

The first four functions are the main SHA-3 functions and are similar to the corresponding functions of the SHA-2 family. They output message digests of 224, 256, 384, and 512 bits, respectively. Similar to the SHA-2 family, those four functions have a collision resistance level that's half the amount of the message digest's size, meaning 112, 128, 192, and 256 bits, respectively, and a preimage resistance level of the same number of bits as the message digest's size, meaning 224, 256, 384 and 512 bits, respectively.

As we can see, these four SHA-3 functions *currently* provide the same level of security as the SHA-2 functions. But attacks on SHA-2 are advancing and this equality may end one day. Attacks on SHA-3 have currently advanced much less than they have on SHA-2.

The SHA-3 Keccak algorithm is slower than SHA-2 because it contains more sequential operations than SHA-2. The SHA3-256 function is approximately 1.5 times slower than SHA-256, while SHA3-512 is 2-4 times slower than SHA-512, depending on the CPU. CPU extensions such as **Intel SHA extensions** and **ARMv8 Cryptography Extensions** support SHA-2, but not SHA-3. However, some CPUs have instruction set extensions that can speed up the process of calculating SHA-3. Examples include **MMX**, **SSE**, **BMI2**, and **AVX2/AVX-512/AVX-512VL** on **x86/x86_64** CPUs, the **SVE/SVE2** and **ARMv8.2-SHA** crypto extension sets on **ARM** CPUs, and **Power ISA** on **POWER8** CPUs. It is worth mentioning that the ARMv8.4-A version of the ARMv8 CPU instruction set also supports SHA-3 acceleration.

As a response to the slowness of SHA-3, the authors of SHA-3 developed the **SHAKE128** and **SHAKE256** functions, which are noticeably faster than other SHA-3 functions. Strictly speaking, they are not hash functions, but **Extendable Output Functions** (**XOFs**). Extendable output functions can generate digests of an arbitrary size and serve as a basis for hash functions. You make a hash function from an XOF by choosing a particular digest size. The numbers in the names of SHAKE128 and SHAKE256 represent the functions' maximum collision attack resistance levels, which are 128 and 256 bits, respectively. Note that due to the birthday paradox, the variable-size digest must be long enough to reach the maximum security level, which is at least 256 bits for SHAKE128 and 512 bits for SHAKE256. SHAKE128 is approximately as fast as SHA-256, while SHAKE256 is 1.5 times slower than SHA-512 on modern x86_64 CPUs.

Later, the same authors presented the **KangarooTwelve** (**K12**) extendable output function, which is based on Keccak but has a reduced the number of rounds (from 24 to 12) and can exploit the availability of parallel execution on multi-core CPUs. The authors claim that KangarooTwelve is approximately 13 times faster than SHA3-256 (0.51 cycles per byte versus 6.4 cycles per byte on an Intel Skylake-X CPU) and still maintains the 128-bit security level. However, the independent benchmarks of real implementations show that KangarooTwelve is only 2-3 times faster than SHA3-256. So far, this security claim is true because the most well-known attack on the SHA-3 algorithm (at the time of writing) can

only attack the reduced 6-round version of Keccak. However, KangarooTwelve has a smaller security margin than the full 24-round SHA-3. As a later development, KangarooTwelve is not considered part of the SHA-3 family and is not supported by OpenSSL.

Even though SHA-3 is slower than SHA-2 in software implementations, there are hardware implementations of SHA-3 that are faster than the hardware implementations of SHA-2 and even SHA-1. And with modern CPUs, the speed of SHA-3 is acceptable.

SHA-3 is currently not nearly as popular as SHA-2. SHA-3 support in cryptographic libraries is already good, but not many applications have started to use hash functions from that family. SHA-3 algorithms are not standardized for use in **TLS** or **X.509** yet and are not in use in popular cryptographic software such as **OpenSSH** and **GnuPG**. A notable use of SHA-3 is the **Ethereum** cryptocurrency, where it is used for proof-of-work checking. It is also worth mentioning that SHA-3 is now undergoing the NIST Post-Quantum Cryptography Standardization process for signature and key exchange. Migration to SHA-3 from SHA-2 is happening quite slowly, similar to how migrating SHA-2 from its predecessor, SHA-1, was.

Reviewing the SHA-1 and SHA-0 hash functions

SHA-1 is a cryptographic hash function that was developed by NSA in the 1990s. The SHA-1 specification was first published in 1993. However, the specification was shortly withdrawn because security flaws were found in the algorithm. The newly revised specification of SHA-1 was published in 1995 and the algorithm from 1993 has the unofficial name of SHA-0.

SHA-1 produces a 160-bit message digest. The algorithm is fast – approximately two times faster than **SHA-256**. **SHA-1** is also approximately as fast as its predecessor, the **MD5** algorithm, and on modern CPUs, it's even faster because of hardware acceleration.

Unfortunately, SHA-1 is not considered secure anymore. SHA-1's security level was originally claimed to be 80 bits, which is not secure by modern standards. The best attack sinks the security level further to 61 bits. In 2017, the first public SHA-1 collision was found. The **Centrum Wiskunde & Informatica** research center from the Netherlands and Google, published two PDF documents with different content but the same SHA-1 hash. If you could obtain an SHA-1 based digital signature of one document, the signature would be valid for the second document as well. This is just one example of why collisions in cryptographic hash functions are dangerous. In 2020, the cost of finding an SHA-1 collision dropped below 50,000 USD, which is easily affordable for large businesses, intelligence agencies, and even rich individuals.

SHA-1 was the most popular cryptographic hash algorithm before SHA-256 took over. It was used in the SSL, TLS, SSH, and IPsec protocols, X.509 certificates, **Pretty Good Privacy** (**PGP**) and **Secure Multipurpose Mail Extension** (**S/MIME**) implementations, **Digital Signature Algorithm** (**DSA**), and other areas. Source control management systems, such as Git and Mercurial, still use SHA-1 for object integrity and identification, though they are continuing to work on migrating to other hash algorithms.

Migration from SHA-1 to SHA-256 was a long process for the IT industry. The famous cryptographer *Bruce Schneier* recommended migrating from SHA-1 in 2005. NIST officially deprecated SHA-1 in 2011 and major web browsers stopped accepting SHA-1-based certificates in 2017. Microsoft stopped support for SHA-1-based signatures for Windows Updates in 2020. Many lesser-known applications have not been migrated yet.

The predecessor of SHA-1 is the MD5 hash function, from the MD hash function family.

Reviewing the MD family of hash functions

The MD family of hash functions consists of four functions: **MD2**, **MD4**, **MD5**, and **MD6**. There are no **MD1** and **MD3** functions. There is also no official information regarding why they don't exist, but rumor has it that MD1 was a proprietary algorithm that was never available to the general public, while MD3 was an experimental algorithm that was abandoned by its designer.

MD stands for **Message Digest**. A fun fact is that **MD5** (as well as **SHA-1** and **SHA-2**) is based on **Merkle-Damgård construction**, which can also be shortened to **MD**.

MD functions were designed by the famous cryptographer Ronald Rivest, the same person who invented **RC** symmetric ciphers, such as **RC2**, **RC4**, and **RC5**, which we reviewed in *Chapter 2*, *Symmetric Encryption and Decryption*.

OpenSSL supports the MD2, MD4, and MD5 hash functions. They all have a message digest size of 128 bits and an initial security level of 64 bits. Unfortunately, those functions are not secure anymore and it's not recommended that you use them. The best attacks reduce the security levels of MD2, MD4, and MD5 to 63, 1 (yes, only one bit!), and 18 bits, respectively. A collision attack on MD4 or MD5 can be performed in a fraction of a second on a typical PC.

MD5 is the most famous function of the MD family. MD5 is a predecessor of SHA-1 and shares a lot of math with SHA-1 and SHA-2. MD5 was very widespread before it was superseded by SHA-1. It was used in all popular cryptographic protocols, standards, and software. It was also very popular for **password hashing** in Unix and web systems. An interesting fact about **MD5** and **SHA-1** is that the **SSL 3.0**, **TLS 1.0**, and **TLS 1.1** protocols used a combination of **MD5** and **SHA-1** because it's hard to find a data block that produces both **MD5** and **SHA-1** collision simultaneously.

The **MD4** function is famous for being used for **password hashing** in Windows NT, 2000, and XP. MD4 has been disabled for password hashing by default since Windows Vista but still can be enabled, even under Windows 10.

Reviewing the BLAKE2 family of hash functions

The **BLAKE2** hash function family consists of several hash functions, two of which are supported by OpenSSL, namely **BLAKE2s** and **BLAKE2b**. **BLAKE2s** produces a 256-bit message digest, while **BLAKE2b** produces a 512-bit message digest. The functions still maintain the maximum collision resistance levels of 128 and 256 bits, respectively. **BLAKE2** functions have a larger security margin than **SHA-2** functions and are similar to **SHA-3** functions.

BLAKE2 functions are famous for their speed. **BLAKE2s** and **BLAKE2b** are faster than **MD5**, **SHA-1**, **SHA-2**, and **SHA-3** on older CPUs without hardware acceleration. On newer CPUs with hardware acceleration, such as Intel Skylake or newer, **SHA-1** is the fastest, but **BLAKE2s** and **BLAKE2b** are still faster than their **SHA-2** counterparts, **SHA-256** and **SHA-512**. **BLAKE2s** and **BLAKE2b** are also noticeably faster than **SHA3-256** and **SHA3-512** while having similar security margins.

The BLAKE2 cryptographic hash function family was developed by an international group of cryptographers in 2012. The BLAKE2 family is based on the BLAKE function family, which was released in 2008, participated in the SHA-3 hash function competition, and even reached the final of the competition, but lost to the Keccak algorithm in the end. The BLAKE function family is based on the **ChaCha** stream cipher, which we reviewed in the previous chapter.

The **BLAKE2** function family already has a successor: the **BLAKE3** function, which was released in 2020. The **BLAKE3** function supports message digests of an arbitrary size that's equal to or longer than 256 bits. **BLAKE3** is a few times faster than **BLAKE2** and can be heavily parallelized because it is based on the **Merkle tree structure**. Regarding BLAKE3 security, the authors claim 128-bit resistance against both collision and preimage attacks. But considerably less security analysis has been done on BLAKE3 than on other hash functions. Hence, it's currently hard to say how secure BLAKE3 is. **BLAKE3** is not included in OpenSSL 3.0.

Even though **BLAKE2** was not standardized for usage in *TLS*, *SSH*, or *PGP*, some software developers noticed the hash function family and decided to use it in their software. Certain BLAKE2 algorithms are used in popular software such as WhatsApp, 7-zip, WinRAR, Rsync, Chef, WireGuard, and cryptocurrencies such as Zcash and NANO.

Reviewing less popular hash functions supported by OpenSSL

OpenSSL also supports other message digest algorithms, but they are not as popular as the ones that we have already reviewed. Some of the less popular algorithms are the national standards of particular countries.

National cryptographic hash functions

GOST94 is a Russian cryptographic hash function that was declassified in 1994. It is based on the **GOST89** block cipher and produces a message digest of 256 bits. GOST94 has an initial security level of 128 bits. There is a theoretical attack that reduces the security level to 105 bits, at the cost of impossible memory requirements. GOST94 has a successor: the **Streebog** hash function, which was published in 2012, also known as **GOST2012** or **GOST12**. Both GOST94 and Streebog are national standards of the Russian Federation.

SM3 is a Chinese cryptographic hash function that was published in 2010. SM3 is similar to SHA-256 in terms of its structure and security. Its security level is 128 bits. SM3 has been a national standard of the People's Republic of China since 2016.

Other cryptographic hash functions

Whirlpool is a cryptographic hash function based on the AES block cipher. Whirlpool was designed by Vincent Rijmen (the co-designer of AES) and Paulo Barreto. The hash function generates a 512-bit message digest and still maintains a collision resistance level of 256 bits. Whirlpool is not a fast hash function; it is approximately 10% slower than SHA3-512. Whirlpool was published in 2000 but has not gained much popularity since then. Some notable uses of Whirlpool include the **TrueCrypt** disk-encryption program and its successor, **VeraCrypt**, which supports Whirlpool as one of its hashing algorithms.

RIPEMD-160 is a hash function that was designed by a group of Belgian cryptographers in 1996, the most popular function from the **RIPEMD family**. RIPEMD-160 is based on the original RIPEMD function from 1992, which, in turn, is based on MD4. RIPEMD-160 generates a 160-bit message digest and has a security level of 80 bits, which is not enough by today's standards. Therefore, it is not recommended to use it.

MDC-2 is a message digest function that uses a symmetric block cipher with a 64-bit block size to produce a 128-bit message digest. MDC-2 stands for **Modification Detection Code 2**. In the OpenSSL implementation, MDC-2 uses DES cipher. Unfortunately, 128-bit digests can have only a 64-bit security level at best, and that is not enough security nowadays. Hence, the MDC-2 function does not conform to modern security standards and is not recommended to use it. It is considered deprecated and is not compiled into OpenSSL by default, even though its source code remains in the library.

Which cryptographic hash function should you choose?

So, which message digest function should you use in your new project? If your project does not have special requirements, choose **SHA3-256**. This function provides good security, is well supported by crypto libraries, has acceptable performance, and is future-proof. Some cryptographers are already advising people to migrate from **SHA-2** to **SHA-3**. It is expected that **SHA-3** functions will get better support in the future versions of security standards and protocols, such as **TLS**, **SSH**, **PGP**, and **X.509**, as well as in popular software that uses cryptography, such as web browsers, PGP and GnuPG, various pieces of digital signing software, and the password hashing subsystems of operating systems and websites. It is also expected that newer CPUs and dedicated cryptography chips will gain better hardware acceleration for **SHA-3** algorithms.

If you need more compatibility and interoperability with existing software here and now, choose **SHA-256** from the **SHA-2** family. But be ready to migrate to an **SHA-3** function if your **SHA-2** security is broken.

If you want more speed and security, and interoperability is not a problem, then choose the **BLAKE2b** function with a 512-bit message digest. It has very good security and is noticeably faster than the **BLAKE2s**, **SHA-2**, and **SHA-3** functions on 64-bit CPUs.

With that, we have learned about the properties of the different cryptographic hash functions supported by OpenSSL, as well as recommendations on what hash function to use in which situation. But enough with theory! Let's do some practical message digest calculation.

How to calculate a message digest on the command line

Calculating a message digest with OpenSSL on the command line is easy. You need to use the `openssl dgst` subcommand of the `openssl` tool. The documentation for this subcommand is available on the `openssl-dgst` man page:

```
$ man openssl-dgst
```

Let's generate a sample file for further message digest calculation:

```
$ seq 20000 >somefile.txt
```

To check which message digest algorithms are supported in your version of OpenSSL, you can use the `-list` switch:

```
$ openssl dgst –list
Supported digests:
-blake2b512              -blake2s256              -md4
-md5                     -md5-sha1                -ripemd
-ripemd160               -rmd160                  -sha1
-sha224                  -sha256                  -sha3-224
-sha3-256                -sha3-384                -sha3-512
-sha384                  -sha512                  -sha512-224
-sha512-256              -shake128                -shake256
-sm3                     -ssl3-md5                -ssl3-sha1
-whirlpool
```

Let's calculate the SHA3-256 digest:

```
$ openssl dgst -sha3-256 somefile.txt
SHA3-256(somefile.txt)= 658656e129914052546af527ba8cf573ab27f-
b47551a0682ffcf00eeaf56d32b
```

As you may recall, message digest calculation is a deterministic operation. No matter how many times we calculate the digest of the same data, the digest will always be the same.

That's how easy it is. Now, let's learn how to calculate the same digest in *C code*.

How to calculate the message digest programmatically

We are going to implement the `digest` program that will calculate SHA3-256 message digest of a file.

Our `digest` program will receive only one command-line argument: the name of the input file to be hashed.

As with symmetric encryption in the previous chapter, let's make a high-level plan for calculating the message digest:

1. First, we must initialize the message digest calculation context.
2. Then, we must feed data to the message digest calculation context chunk by chunk, reading chunks from the input file.
3. Next, we must finalize the calculation and get the message digest.
4. Finally, we must print the calculated message digest to *stdout*.

Similar to symmetric encryption, OpenSSL contains the new **EVP API** and the old, deprecated low-level API for message digest calculation. The functions of the EVP API start with the `EVP_` prefix, while the functions of the low-level API start with hash function-specific prefixes, such as `MD5_` or `SHA256_`. We discussed the disadvantages of the old, deprecated API in the previous chapter, so we will not discuss this again here. As with symmetric ciphers, we are going to use the EVP API for message digest calculation. Even if we wanted to use the low-level API functions, we wouldn't be able to use them for SHA3-256, since SHA-3 algorithms are relatively new and are not supported by the old API.

Implementing the digest program

Let's implement our `digest` program:

1. First, we must create and initialize the message digest calculation context:

```
EVP_MD_CTX* ctx = EVP_MD_CTX_new();
EVP_DigestInit(ctx, EVP_sha3_256());
```

2. Then, we must feed the data from the input file into the context:

```
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_
file);
    EVP_DigestUpdate(ctx, in_buf, in_nbytes);
}
```

3. Once all the data has been fed to the context, we must finalize the calculation and get the message digest into the `unsigned char` array:

```
const size_t DIGEST_LENGTH = 256 / 8;
unsigned char md[DIGEST_LENGTH];
EVP_DigestFinal(ctx, md, NULL);
```

4. Once we have obtained the message digest, we don't need the `EVP_MD_CTX` object anymore. We have to free it to avoid memory leaks:

```
EVP_MD_CTX_free(ctx);
```

5. The only thing left to do is print the calculated message digest. Let's print it in the same format that the `openssl dgst` subcommand does:

```
printf("SHA3-256(%s)= ", in_fname);
for (size_t i = 0; i < DIGEST_LENGTH; ++i) {
    printf("%02x", md[i]);
}
printf("\n");
```

Compared to symmetric encryption, message digest calculation does not need many parameters, such as keys, initialization vectors, or padding types. Cryptographic hashing only needs a hash function type and input data. There are no requirements about input data format, any bitstream can be hashed. Thus, it is difficult to fail to calculate a message digest. If your program experiences failures with hashing, the most probable reasons are input/output errors or initialization errors, such as a particular hash function not being compiled into your version of OpenSSL, which means that the message digest's calculation cannot be initialized.

The full source code for the `digest` program can be found on GitHub in the `digest.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter03/digest.c`.

Running the digest program

Let's run our digest calculation program on the same `somefile.txt` file that we used in the previous section:

```
$ ./digest somefile.txt
SHA3-256(somefile.txt)= 658656e129914052546af527ba8cf573ab27f-
b47551a0682ffcf00eeaf56d32b
```

As we can see, that calculated digest is identical to the digest that was calculated by the `openssl dgst` subcommand. This confirms that our program works as expected.

Summary

In this chapter, we learned what cryptographic hash functions and message digests are. We also learned about the common attacks that are performed on hash functions and how to assess a hash function's security. Then, we reviewed the cryptographic hash functions supported by OpenSSL and how much security they provide. We finished our review by providing a practical piece of advice on which hash functions to use in which situations.

In the practical section, we learned how to calculate a message digest using the `openssl dgst` subcommand. Finally, we learned how to develop a C program that will use the OpenSSL library for message digest calculation. We compared a message digest that was generated by our program with one that was generated by `openssl dgst` and confirmed that our digest was generated correctly.

In the next chapter, we will learn about **Message Authentication Codes** (**MACs**) and **Hash-based Message Authentication Codes** (**HMACs**). HMACs are based on cryptographic hash functions, which is one of the reasons why it was important to learn about hash functions and message digests in this chapter.